Quickly finding near-optimal storage designs

ERIC ANDERSON, SUSAN SPENCE and RAM SWAMINATHAN
HP Labs
and
MAHESH KALLAHALLA
DoCoMo Labs USA
and
QIAN WANG
Pennsylvania State University

Despite the importance of storage in enterprise computer systems, there are few adequate tools to design and configure a storage system to meet application data requirements efficiently. Storage system design involves choosing the disk arrays to use, setting the configuration options on those arrays and determining an efficient mapping of application data onto the configured system. This is a complex process because of the multitude of disk array configuration options, and the need to take into account both capacity and potentially contending I/O performance demands when placing the data. Thus, both existing tools and administrators using rules of thumb, often generate designs that are of poor quality.

This paper presents the *Disk Array Designer* (DAD), which is a tool that can be used both to guide administrators in their design decisions and to automate the design process. DAD uses a generalized best-fit bin packing heuristic with randomization and backtracking to search efficiently through the huge number of possible design choices. It makes decisions using device models that estimate storage system performance. We evaluate DAD's designs based on traces from a variety of database, filesystem, and e-mail workloads. We show that DAD can handle the difficult task of configuring mid-range and high-end disk arrays, even with complex real-world workloads. We also show that DAD quickly generates near-optimal storage system designs, improving in both speed and quality over previous tools.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management

General Terms: Algorithms, Design, Management, Performance

1. INTRODUCTION

Businesses rely critically on the data stored in their enterprise computer systems, so the onus is on storage administrators to ensure that the storage can support applications' data capacity and I/O performance requirements. However, designing these storage systems is a challenge. For many enterprise applications, the storage system consists of many terabytes of disk space, spread across multiple disk arrays. Administrators must make decisions about what types of storage devices to use, how to configure them, and where to place the application data. These issues are complicated because there are a large number of storage devices to choose from, many configuration options for these devices, and a complex dependency between performance and data layout. To make matters worse, the scale and complexity involved in making these decisions continues to increase as storage systems and application workloads grow, and as the storage devices provide richer feature sets.

Modern enterprise class storage systems are built from disk arrays, which provide a huge number of configuration options. While these disk arrays may seem simple to configure at first glance, in reality administrators must carefully set or choose many of the options before any data can be stored on the arrays. Examples of such options are the number and type of disk drives, the RAID configuration, degree of redundancy in communication paths between disk controllers and disks, and the way the array is partitioned into logical storage areas. In addition, the options themselves depend on the architecture of the disk array; so they are different for different arrays, and sometimes even for different generations of the same array. In order to make disk arrays suitable for a wide variety of applications, array vendors leave most of the configuration options as tunable, with only loose guidelines on how to set them.

Many enterprise applications have demanding requirements on storage: substantial data storage capacity, high-performance I/O and high availability. Designing a storage system configuration to meet all these requirements is complex. The application data is managed by filesystems or databases as files or records within large, logically-contiguous regions of storage known as logical volumes. Part of the storage design problem is determining the placement of such regions in the storage system. Doing this in a way that meets just the capacity requirements is already a hard problem; taking into account the I/O performance requirements as well makes it even harder. The biggest challenge is to meet the requirements of multiple applications that use the same storage system. The current trend in enterprise IT is to exploit the business benefits of the shared resource pool that results from consolidating data for multiple applications into a single storage system. However, because of the resulting lack of isolation, storage configuration design must take into account the interactions between applications while making data placement decisions.

Storage systems can represent a significant fraction of an enterprise's IT capital budget (a single high-end storage system can cost more than one million dollars) so incorrect provisioning can be extremely expensive. Today, storage systems are typically designed by hand using rules of thumb. This approach has two problems. First, it takes a lot of human effort, as it requires skilled and expensive administrators (the current estimate is that it takes one administrator for every 54TB of storage [Hewlett-Packard Development Company 2003]) and the systems cost 3-4 times more to manage than to purchase [Morris 2002; Allen 2001; Lamb 2001]. Second, due to the complexity and scale of enterprise storage systems, designs generated by even skilled administrators may be poor. The resulting systems may be massively over-provisioned (factors of 3–4 are commonplace), which makes them unnecessarily expensive. If, alternatively, they end up being under-provisioned, they perform poorly, hindering applications and users.

To counter the complexities, application and device vendors recommend simple rules of thumb for administrators to follow. However these rules provide no guidance as to how many resources are required and are not always applicable. For instance, Oracle now recommends the SAME (Stripe And Mirror Everywhere) [Loaiza 2002] approach. However, this approach requires mirroring, which uses 1.5-2 times more disk space than RAID 5. While many users are reassured by the use of mirroring, there are some even high end database users who do prefer RAID 5 e.g. for data warehousing applications, decision support databases or clones of production databases [Uysal 2004]. Moreover, SAME still doesn't provide guidance on how many resources are needed to meet the workload's requirements.

Ideally, administrators should have the tools to enable them to both determine quickly if they have under or over-provisioned an existing storage system and to enable them to obtain new storage designs that are "near-optimal" (with respect to whatever metric they choose) in meeting application requirements. This paper presents a *Disk Array Designer* tool that meets both these needs.

The Disk Array Designer (DAD) generates a *storage system design* based on an input workload, a set of storage device types it can use, and a user-specified goal, such as a minimal system price or balanced design. The basic structure of DAD is illustrated in Figure 1. DAD aims to generate a placement of data onto a storage configuration that meets the workload's requirements, while ensuring the placement is near-optimal with respect to the user's goal. Thus, if the goal is minimal price, the data will be packed onto the minimum number of disks and array components that will still meet the workload's performance requirements. The generated design captures both the configuration of the disk arrays in the storage system and the mapping of regions of application data onto the storage system.

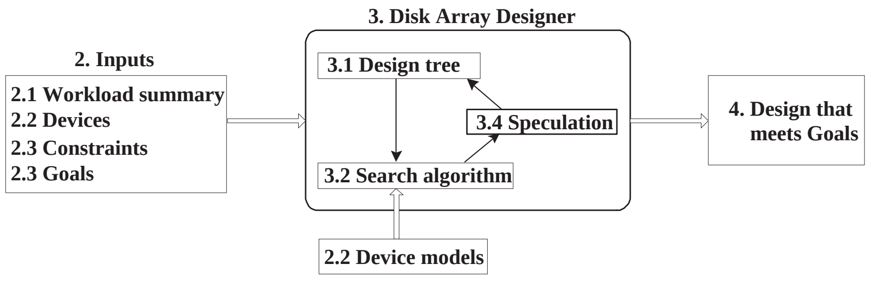


Fig. 1. The structure of DAD, including its inputs and outputs. The numbers correspond to the sections of the paper in which aspects of DAD and its inputs and outputs are described.

Administrators can use DAD in a number of ways. They can compare DAD designs to the storage configuration they are currently using, enabling them to determine quickly if they have under or over-provisioned their storage system. They can also take advantage of DAD's fast runtime and flexibility of use to generate valid solutions that answer "what if" questions. This allows them to experiment with making changes to specific parts of the storage system. Examples of such "what if" questions include: "What is the best design if I reconfigure this array, place this data in this way, or restrict the configurations of placements in various ways?" Being able to find out the answer to the latter question, for example, is particularly useful when considering changes to an existing system, where the administrator wants to determine the optimum placement for new data, without moving existing data.

An important use for DAD is its role in an automated storage management system such as Hippodrome [Anderson et al. 2002]. Hippodrome iterates through a loop, first generating (or improving) the storage system design, second implementing the new design, and third analyzing the workload as it runs on the new storage system. Hippodrome's automated approach can reduce system management costs, which is significant given that studies [Allen 2001; Lamb 2001] have indicated that the cost of a large storage system, over the course of its lifetime, is dominated by the cost of management. Only with a tool like DAD to automate storage system design, is it possible to take an automated approach to storage management.

Although other solutions have been proposed for storage system design, they have the following two deficiencies: a) they do not handle the scale and complexity of today's applications and disk arrays and b) they are not fast or flexible enough to be used for "what if" analysis and automated system management tasks. For instance, although Minerva [Alvarez et al. 2001] does address the same basic problem as DAD, it is not designed to handle the sophisticated configuration requirements of mid-range and high-end disk arrays. We provide a detailed comparison with Minerva in Section 4.4, and other related work in Section 5.

The storage design problem can be viewed as an optimization problem, where the aim is to find the design that best meets the capacity and I/O performance requirements of a set of workloads. We analyze the challenges of this problem and explain DAD's approach to solving it. The capacity only version of the problem using identical sized disks is the NP-complete bin-packing problem [Garey and Johnson 1979]. However, existing bin-packing heuristics [Hill 1994; Johnson et al. 1974a; Shor 1986; Kenyon 1997] can't be directly applied to storage system design because of the extra constraints introduced by I/O performance requirements. Whereas the existing bin-packing heuristics assume that constraints are additive, the device utilization constraints in storage systems (quantifying I/O performance) are non-additive; i.e., the aggregate utilization of a device when two data items are placed on the same device is not the same as the sum of the utilizations of the device when they are individually placed on the device. Therefore, DAD generalizes best-fit bin-packing to accommodate the constraints specific to storage systems. To achieve this, first we use randomization as it usually helps in complex search problems. Second, we use reassignment, where existing parts of the assignment are undone and then retried, to help avoid local minima. Third, we run reassignment at different granularities to fine-tune the design, similar to the way in which simulated annealing [Metropolis et al. 1953] operates at different temperatures.

Because the storage design problem is NP-Hard, DAD is designed to find near-optimal (as opposed to optimal) solutions in a reasonable amount of time. We provide experimental evidence to show that the solutions generated by our design tool are near optimal for real-world workloads. We keep the runtime within the range of minutes to a few hours for a full design of a large storage system, and shorter than that for minor revisions to the storage system. This allows us to use DAD both for automatic adaptation of a storage system design and for providing designs to administrators.

The primary contribution of this paper is to describe DAD, a fast tool for finding near-optimal storage system designs. Section 2 summarizes the inputs and dependencies of this storage system design tool. Section 3 describes the architecture, the central data structure, and the search techniques used in DAD. Section 4 experimentally evaluates DAD. Section 5 provides a summary of other related work, and finally, Section 6 concludes and discusses future directions.

2. INPUTS TO THE DISK ARRAY DESIGNER

The inputs to DAD are a workload description, a set of potential devices, optional user-specified constraints, and a user-specified goal; we will detail these inputs in this section. DAD chooses the appropriate types of devices to use, the configuration of each device and the mapping of application data onto the devices, to best achieve the specified goal.

2.1 Workloads

The workload description, using the language presented in [Wilkes 2001] describes capacity and performance requirements in terms of *stores* and *streams*.

A *store* is a logically contiguous block of storage used to hold some application data, for example, a file system or a database table-space. The most significant attribute of a store is its size. Stores are packed into *logical units* (LUs), and these LUs map to disks that are grouped together using RAID [Patterson et al. 1988] on configured disk arrays.

A *stream* summarizes an I/O request pattern. Each stream is associated with one store. A stream is characterized using the following attributes:

—**requestRate**: the mean of the number of I/O requests per second (IOPS) sent to the store;

—**requestSize**: the mean of the number of bytes read or written by a single request;

—**runCount**: the mean of the number of consecutive I/O requests issued to contiguous addresses;

—**queueLength**: the mean number of I/Os queued on the store;

—**onTime**: given that streams may be active for a while and then inactive for a while, the mean of the number of seconds for which a stream is generating I/Os to a store;

—**offTime**: the mean of the number of seconds for which a stream is not generating I/Os;

—**overlapFraction**: when two streams have correlated onTimes, the fraction of the onTime of this stream for which another stream is also active. (It has a value of 1 when both are active for the same time.)

The store and stream attributes are specified as part of DAD's run-time configuration file. Figure 2 provides examples of a store and a stream as they would be specified in that file. Further information on stream attributes can be found in prior work on workload modeling and characterization [Wilkes 2001; Alvarez et al. 2001; Uysal et al. 2001; Merchant and Alvarez 2001].

2.2 Devices and performance models

Storage devices present two kinds of constraints to the storage system designer: *Configuration constraints* that specify restrictions on the physical configuration of the array components, and *Performance constraints* that specify limitations on the performance achievable with a certain configuration and mapping of data to those devices.

The architecture of disk arrays poses fundamental constraints on possible configurations. These constraints include limits on the number of disks in a RAID group, the maximum number of disks in the device, and the number of controllers in the disk array. These constraints are hard-coded into DAD for each type of storage device, so that DAD can only generate valid configurations.

DAD uses device performance models to evaluate the designs that it generates. The device performance models take a possible device configuration and a mapping of stores and streams onto the LUs of a device, and return a prediction of the utilization of the components of the device. Consider this simple example: if a single disk can handle 100 small randomly-placed reads/second, and it has two uncorrelated streams on it each performing 25 small randomly-placed reads/second, then the performance model would

```
store store_1 {
    { capacity 7.986e9}          # A 7GB store
    { boundTo FC60_0001.2 }      # mapped to a logical unit
};                               # on an FC-60 array (not shown)

stream stream_3 {                # A stream
    { boundTo store_1 }          # bound to that store
    { requestRate {
        { all 9.478}             # Overall request rate
        { read 4.740}            # rate for reads
        { write 4.738}           # and writes
    }}
    { requestSize {              # Request size in bytes
        { all 52444.933}
        { read 52435.252}
        { write 52454.619}
    }}
    { runCount 1.844}            # Avg 1.8 consecutive I/Os
                                 #   to contiguous addresses
    { onTime 33.253}             # Seconds for which stream does/
    { offTime 83.500}            #   doesn't generate I/Os to store
    { queueLength 2.377 }        # 2.377 mean I/Os queued at device while
 # the stream is on.
    { overlapFraction {          # Fraction of ontime for which:
        { stream_1 0.509}        #   stream_1 overlaps with stream_3,
        { stream_2 0.054}        #   stream_2 overlaps with stream_3,
        { stream_4 1.000}        #   stream_4 overlaps with stream_3, ...
        { stream_5 1.000}
        { stream_6 0.002}
    }}
};
```

Fig. 2. A (much simplified) workload specification sample. One store is accessed by one stream. For simplicity, the datamodels shown are simple numeric values but, in practice, distributions would normally be used.

report that the disk is 50% utilized. Note that the actual models are much more complex than this simple example would suggest, because they take more factors into account.

DAD needs device performance models that can be evaluated quickly, so that they can be used for exploring a vast number of possible design choices in a short period of time. Analytic device performance models meet this requirement; they use analytical formulae to calculate the utilization quickly. In comparison, DAD would not be able to use an I/O trace with a detailed simulator because this would be too slow, thereby limiting the size of the designs that can be generated in any reasonable amount of time.

In principle, DAD can be used to configure any disk array as long as good analytical models are available. However, for the purposes of this paper, we restrict our focus to two HP disk arrays for which we have validated models: the FC-30 [Hewlett-Packard Company 1998], a low-end 30-disk array, and the FC-60 [Hewlett-Packard Company 2000], a mid-range 60-disk array. An illustration of the FC-60 is shown in Figure 3. We chose these arrays because accurate analytical models were already available for them. Methods for generating performance models for arbitrary disk arrays are presented in [Uysal et al. 2001] and [Anderson 2001], but have been validated only for the above two arrays.

The three different performance models that DAD uses are: *monolithic equation-based*

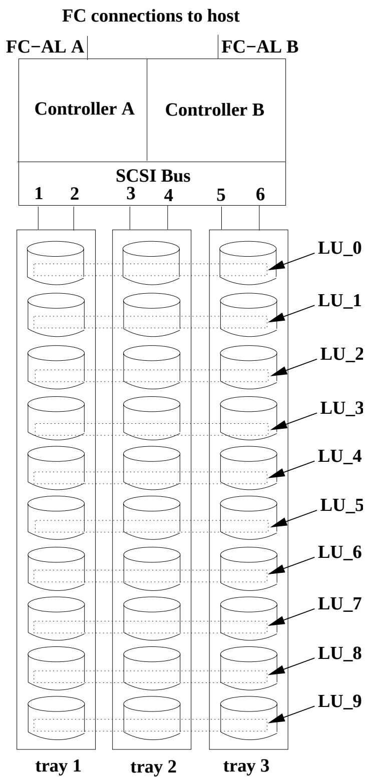


Fig. 3. One configuration of an FC-60 disk array.

models as described in [Merchant and Alvarez 2001], *modular equation-based models* as described in [Uysal et al. 2001], and *table-based models* as described in [Anderson 2001], depending on which device types are used. All the models use summaries [Wilkes 2001; Veitch and Keeton 2003] of the workload (I/O traces) to make their predictions.

The *monolithic equation-based models* treat the FC-30 disk array as a controller model plus a RAID group model. Both the controller and RAID group models are queuing models with a calibration factor applied to correct the performance estimates of the base queuing model. The monolithic equation-based models do not use the queue-length attribute of streams. The *modular equation-based models* treat the FC-60 disk array as a set of modular components. They have a model of an individual disk, an individual controller and an individual bus. The RAID group model composes together disk models to calculate RAID group utilization, and transforms the input workload description from the workload applied to the entire group to the workload applied to the disks in the RAID group. The array model composes all of these sub-models together to calculate the overall utilization. The *table-based models* calculate the interference between streams as done by the modular equation-based RAID group model, determine individual stream utilization on a component by performing an interpolation on a large table of measured values, and then compose the individual utilizations together using the same calculation as in the modular equation-based models.

DAD treats the models as black boxes that take in possible designs and return storage device component utilizations, so it is not necessary to understand the models in order to understand the mechanics of DAD. Some of the models can calculate utilizations more quickly if a new design is similar to a previously checked design. DAD uses this fact to improve its performance, but the quality of the design is not dependent on the type of model.

The real-world quality of solutions generated by DAD does depend on the accuracy of the models. DAD will find a solution that is near-optimal with respect to user goals, relative to the accuracy of the models used. The use of DAD as the solver for Hippodrome [Anderson et al. 2002] on synthetic and file-system benchmarks, earlier work in Minerva [Alvarez et al. 2001] on a TPC-D-like workload, and the work on storage-system modeling [Shriver 1997; Anderson 2001; Uysal et al. 2001; Merchant and Alvarez 2001] demonstrates that models can predict workload performance with sufficient accuracy to support a workload's requirements.

2.3 User specified constraints and goals

DAD users may place additional constraints on valid designs, by specifying them as part of the input to DAD. The following constraints can be specified: the available disk array types; restrictions on the valid device configurations, by limiting choices of RAID levels or disk types; a maximum total system price or a maximum utilization of any device; and restrictions that certain stores be placed on particular devices or on LUs with particular RAID levels.

The final input is a goal, which is used by the design tool to determine the "best" choice from a set of possible configurations. There are many possible goals for a storage system design, such as minimal price, balanced load or low utilization. The goal is implemented as a comparison function over two different configuration choices. These comparison functions are hard-coded into DAD. Currently, DAD provides a library of 42 comparison functions and a generic interface for the definition of additional functions. The existing library includes functions for minimizing price, balancing load, and minimizing average device utilization. The comparison function to be used is specified at run-time. Throughout this paper, we will use the term "price" to indicate dollar amounts and "cost" to indicate the target of the arbitrary user specified goal function.

3. ARCHITECTURE OF THE DISK ARRAY DESIGNER

DAD consists of three main parts: a data structure, called the *design DAG* (directed acyclic graph), that keeps track of the current design and some potential alternatives; the search algorithm that explores the space of possible designs; and a state management component, called *speculation* that, during the search, allows DAD to efficiently switch to a previously generated design when needed.

3.1 Design DAG

The design DAG represents the design of a storage system. The root of the design DAG contains references to *device DAGs* for each storage device in the system. A device DAG is an abstraction of the architecture of a disk array. It is a directed acyclic graph with two kinds of nodes: components and attributes. Given an architectural specification of a disk array, a device DAG can be derived by mapping parts of the disk array, such as controllers and LUs, to components, and properties of the components, such as capacity

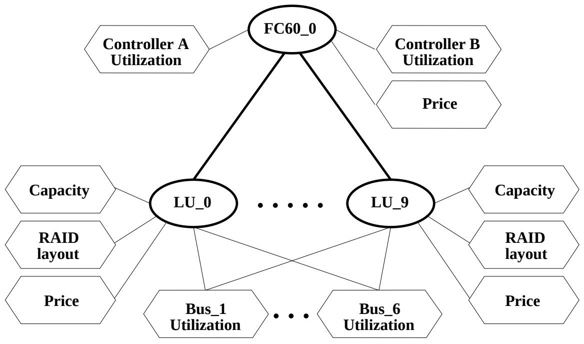


Fig. 4. The abstract representation of an FC-60 disk array as a device DAG. The disk array and its component parts are represented as ovals, while attributes of each component are represented by hexagons.

and utilization, to attributes. An edge from a component C to an attribute A indicates that A is an attribute of the component C. An edge from component C to component D indicates that C is on the data path between the host and component D.

A device DAG does not contain components and attributes for every part and property of a disk array; it only includes the details that are relevant to designing storage systems — those for which we have performance models or for which DAD can consider alternative configuration options.

The device DAGs, for the modern disk arrays we have considered, are acyclic because their architecture is hierarchical. The subgraph constructed from component nodes is a tree for all the disk arrays we have encountered. Also, attribute nodes are normally associated with exactly one component node – the exceptions are cases where an attribute is related to multiple components, such as the utilization of a SCSI bus shared by multiple logical units. The number of levels in the device DAG is determined by the complexity of the array type: mid-range arrays have two levels, while high end arrays have three or more. As we shall discuss later (Section 3.2), this hierarchical structure is exploited by our search algorithm.

Next, we will illustrate the correspondence between the architecture of a disk array and its representation in a device DAG. To demonstrate how DAD represents different disk arrays, we provide examples of the device DAGs for a mid-range array and a high-end array.

The architecture of the mid-range FC-60 disk array is presented in Figure 3 and the device DAG derived from it is presented in Figure 4. As shown in Figure 3, an FC-60 has two front end controller modules A and B, each of which is attached to the six shared SCSI buses, which in turn connect to the disks. In the configuration shown in this example, the disk array has been configured for high availability, with redundant SCSI bus connections to each of the three enclosures (trays) containing the disks. There are ten disks in each tray. Three-disk LUs are created on the disk array, under the control of a logical volume manager (LVM), and presented to the hosts, providing 10 LUs altogether. Each LU spans multiple SCSI buses for performance and availability. Hosts connect to the disk array via a fibre-channel arbitrated loop (FC-AL), through the two ports to the two controllers.

Figure 4 shows the corresponding device DAG. The root of this device DAG represents the entire disk array. Our FC-60 performance models compute the utilization of the two controllers, so attribute nodes representing the controller utilizations are associated with

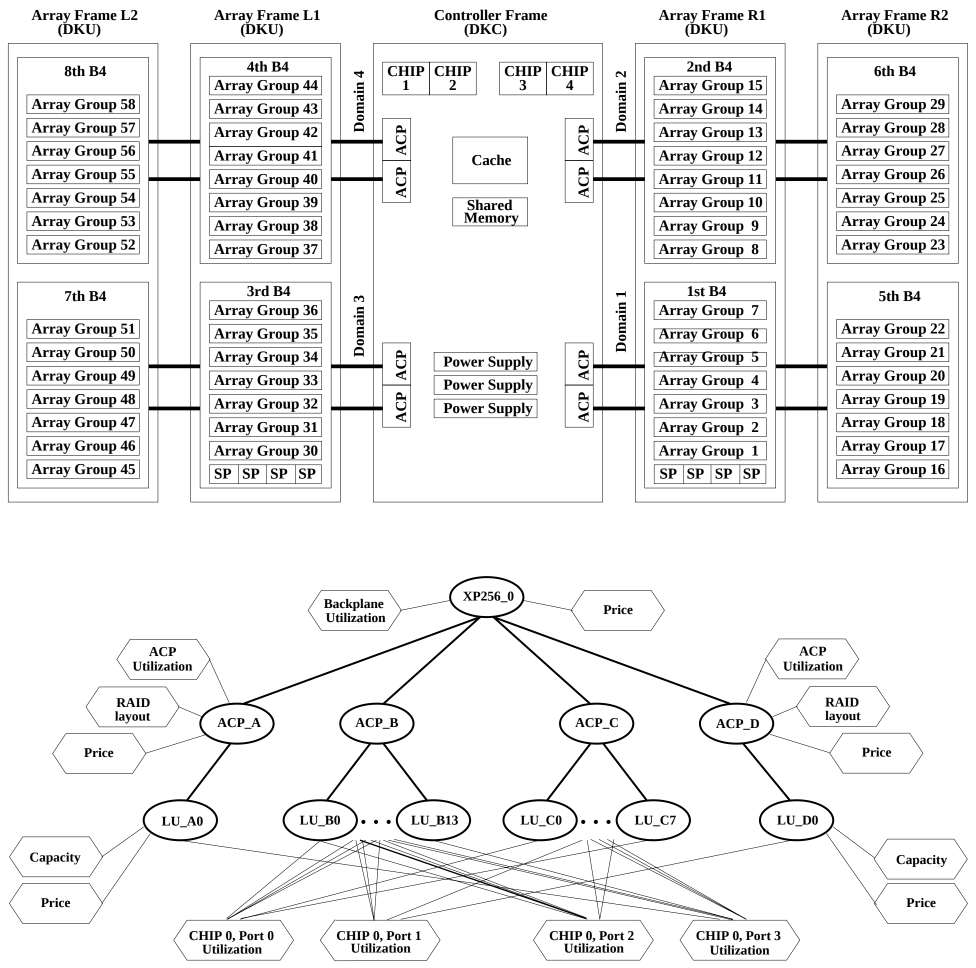


Fig. 5. A configuration of an XP-256 disk array and the corresponding XP-256 device DAG.

the device DAG root. The component nodes for the LUs are directly attached to the root and they have attribute nodes such capacity and SCSI bus utilization. SCSI bus utilization is also calculated by our performance models. Because SCSI buses may be shared by multiple LUs, a SCSI bus's utilization is an attribute of all the LUs that share that SCSI bus; hence the SCSI bus utilization attribute nodes that are shown have edges from multiple LUs.

Now, consider the architecture of the high-end XP-256 disk array and the device DAG derived from it in Figure 5. As shown in the disk array configuration [Hewlett-Packard Company 2001], an XP-256 has a control frame with two disk frames on either side. The contents of the control frame include four fibre-channel chips, with four ports per chip, and four Array Controller Processor (ACP) pairs that can be attached to the backplane. The disk frames are enclosures containing disks arranged in Array Groups, presented to the host as LUs.

The structure of the corresponding device DAG reflects the multi-level internal structure

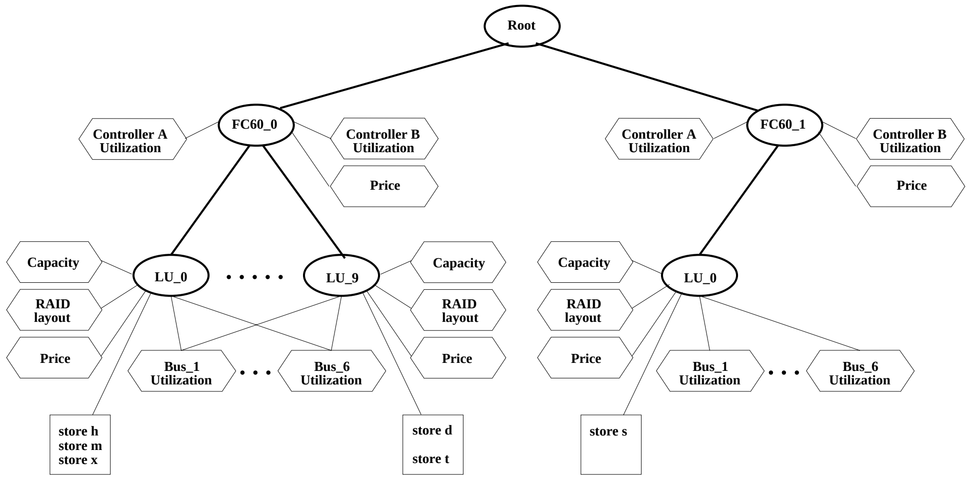


Fig. 6. A design DAG during configuration design. The DAG contains one fully-configured FC-60 and a second FC-60 that has just been added to accommodate one more store.

of this disk array. The root of the device DAG represents the entire disk array, component nodes for the ACPs are directly attached to the root and each LU is attached to an ACP. Our XP-256 performance models compute the utilization of the XP-256's Backplane, each of the ACP pairs and the ports of the fibre-channel chips, so attribute nodes representing these utilization properties are associated with the device DAG root, ACPs and LUs respectively. The edges from multiple LUs to an attribute node for port utilization on a fibre-channel chip reflect the fact that multiple LUs share that port.

3.1.1 *Using the device DAG.* Instances of the device DAG are built up incrementally during the execution of DAD's storage system design algorithm. Each instance represents the configuration of a disk array used in the storage design, including the LUs created on the disk array under the control of a logical volume manager, the mapping of stores to those LUs, and the performance demands that such data placement puts on various disk array components. As the algorithm executes, nodes in the device DAG get instantiated, stores are assigned and attribute values are adjusted accordingly. Figure 6 shows how instances of the FC-60 device DAG illustrated in Figure 4 are incorporated into a design. In this design, LUs are instantiated as 3 disk LUs, stores are shown assigned to the LUs, the capacity used by stores is represented by the capacity attribute attached to the LU component and the utilization of components by I/O streams accessing those stores is represented by the utilization attributes attached to the disk array and LU components.

At run-time, DAD's search algorithm incrementally builds up and explores the component nodes of a design. Note that these nodes represent components that can be added, removed or transformed during the search. Component addition, removal and transformation functions are specified and associated with specific components, for each array type supported by DAD. These functions are called by DAD to determine how the component and its subcomponents can be added, removed, or transformed. For instance, addition functions associated with the FC-60 disk array component indicate where new LU components can be added to the current design, and transformation functions associated with the same

FC-60 disk array component indicate where an existing LU can be changed from a RAID 1/0 LU into a RAID 5 LU.

Disk array configuration options are encoded in the components, and may be limited by constraints input to DAD at run-time (as described in Section 2.3). DAD uses this information during the execution of its search algorithm to determine what configuration changes can be made to the current design. Thus, for instance, constraints on the available disk array types inform the creation and addition of new disk arrays to the root of the design DAG, and constraints on disk types and RAID levels for LUs of a particular disk array type inform the addition and transformation functions for components associated with that disk array type.

At run-time, DAD uses attributes attached to components in the device DAG in two slightly different ways. It uses some attributes to encapsulate the device performance models: Figure 4 shows the device performance models for the FC-60 encapsulated in the controller utilization and SCSI bus utilization attributes. It uses other attributes to hold information, such as price and capacity, that are used to aid decisions about store assignment into a design DAG, based on the user's specified goal function.

Section 3.2, below, provides more details on how the search algorithm uses attributes to inform its decisions about disk array configuration and store assignment.

3.2 Overall search algorithm

DAD's storage design algorithm works in three phases: *initial assignment*, *batched reassignment* and *single store reassignment*. The aim of initial assignment is to generate a design that meets the workload's requirements. The aim of batched reassignment is to tune the design to meet user goals more optimally by removing stores, eliminating devices and trying to fit the stores elsewhere. The aim of single store reassignment is to fine-tune the design by moving stores individually.

The storage design algorithm is shown in Algorithm 1. All three phases use a single store assignment algorithm, described in Section 3.3, to add a single store to the current set of device DAGs, instantiating new device DAGs if necessary.

Ideally, the reassignment rounds would not be necessary and the initial assignment would be optimal. However, this is not the case for DAD mainly because it is very difficult to determine the right order in which to add stores to give the best configuration. Though heuristics such as ordering by size have been shown to be appropriate for traditional bin packing problems [Coffman, Jr. et al. 1997], we believe that these heuristics are not appropriate for the search problem in DAD: this is mainly because devices have multiple attributes and the cost function is not linear. This intuition is justified experimentally, in Section 4.3, where we present evidence that even repeated initial assignment is insufficient to produce good results on realistic workloads.

The user specifies their goals to DAD by specifying the names of comparison functions to use; these names are specified in DAD's run-time configuration file. There are two types of comparison functions: *path comparison functions* and *design comparison functions*. Both functions select between alternative designs. The path comparison functions are used when assigning a single store, and compare between two possible assignments of a store into a DAG. Since any store assignment is along a single path, from the root of the DAG to the leaf node at which assignment is being done, it is sufficient, and most efficient, to compare just that path. The design comparison function compares two complete designs. The user specifies a path comparison function for initial assignment, batched re-

Algorithm 1 DAD's design space search algorithm. The first phase generates a valid initial design. The second phase optimizes this design at a broad level to better meet user goals, eliminating devices if possible. The third phase optimizes the resulting design at a fine level, to generate a more balanced configuration based on the user goals. The Assign_Stores_With_Tree_Expansion procedure instantiates new device DAGs.

```
 1: procedure Assign_Stores(store_list $S$, user_goals $G$,
                            batch_reassignment_rounds $n$,
                            leaves_reassigned_per_round $r$,
                            store_reassignment_rounds $m$)
 2:   Initialize device DAGs based on user specification
 3:   {Initial assignment of all stores in $S$}
 4:   Randomize the order of stores list $S$
 5:   Assign_Stores_With_Tree_Expansion($S$, $G$)
 6:   {Batched reassignment with $n$ rounds and $r$ leaves per try}
 7:   for batch_reassignment_round = $1 \ldots n$ do
 8:     while All leaves of all device DAGs have not been processed in this round do
 9:       Remember current storage design $D$
10:       $R$ = set of $r$ leaves not previously selected in this round
11:       $S$ = set of all stores on the leaves in list $R$
12:       for every store $s$ in list $S$ do
13:         Delete $s$ from its current device DAG
14:       Delete any sub-trees that have no stores assigned to them
15:       Randomize the order of stores list $S$
16:       Assign_Stores_With_Tree_Expansion($S$, $G$)
17:       Comparing current design and previous design $D$, choose best based on user goals $G$
18:   {single store reassignment with $m$ rounds}
19:   for store_reassignment_round = $1 \ldots m$ do
20:     Remember current storage design $D$
21:     $s$ = a randomly chosen store from set of all stores
22:     Delete $s$ from its current leaf node
23:     Assign_Stores_With_Tree_Expansion($\{s\}$, $G$)
24:     Comparing current design and previous design $D$, choose best based on user goals $G$

25:   procedure Assign_Stores_With_Tree_Expansion(store_list $S$, user_goals $G$)
26:     for store $s$ in list $S$ do
27:       path $B$ { the best path found for assigning store s into a tree }
28:       for every instantiated device DAG $d$ do
29:         Assign_Single_Store($s$, root($d$), $B$, $G$)
30:       for all disk array types $t$ do
31:         Instantiate a new device DAG $d$ of type $t$
32:         Assign_Single_Store($s$, root($d$), $B$, $G$)
33:       if no assignment was found, emit an error, this store cannot be assigned to any device
```

assignment and single store reassignment. They also specify a design comparison function for use in batched reassignment and single store reassignment. DAD currently implements 42 path comparison functions and 4 design comparison functions; it has a generic interface for programming additional path and design comparison functions. For example, one of the existing comparison functions supports the goal of minimizing the price of the storage system while evenly utilizing the devices. We evaluate a subset of these functions in Section 4.

Both the initial assignment and reassignment phases use a common subroutine *Assign_Stores_With_Tree_Expansion (store_list, user-goals)* (Algorithm 1) to find the best assignment for each store in the design, potentially adding devices and nodes to the design. This function uses the subroutine *Assign_Single_Store (store, node, best-path-so-far, user-goals)* (Algorithm 2). Assign_Single_Store starts from the given node, that represents a component of a device DAG, and tries to find a better way to incrementally add a single store into the DAG rooted at this node. We defer the details of this function to the next section (Section 3.3). Lines 25–33 of Algorithm 1 show how the *Assign_Stores_With_Tree_Expansion (store_list, user-goals)* subroutine performs store assignment on the design DAG. First, an attempt is made to assign the store to each of the existing device DAGs in the design. Next, the search continues by instantiating a new device DAG of each available disk array type in turn. This function finds the best assignment of the store across all of the device DAGs; useful when user goals aim to make a price–performance trade-off. Finally, if no assignment is possible, the procedure generates an error.

Lines 2–5 of Algorithm 1 show the initial assignment phase of the DAD search algorithm. First, an initial set of device DAGs are initialized, based on any user specification; in particular, if the user has specified any pre-existing devices and their configuration, the corresponding device DAGs are instantiated here. Then, the list of stores is randomized to avoid any bias toward a particular design or goal. Experiments have shown that the ordering of stores during initial assignment ultimately has little effect on the configurations generated after reassignment. Intuitively, this is because the bin-packing problem has multiple dimensions and there is no natural sorting order. Finally, the initial design is generated using the *Assign_Single_Store_With_Tree_Expansion* function, which adds each store, one by one, into the design.

Lines 6–17 of Algorithm 1 show the batched reassignment phase of the DAD algorithm. In this phase, DAD tries to find a better packing that uses fewer LUs, by performing leaf level reassignment. Recall that all LUs are leaves in the device DAG. Batched reassignment is performed for *n* full rounds, where each full round of reassignment is composed of multiple sub-rounds. In each sub-round, the stores belonging to *r* leaf nodes are reassigned. The *r* leaves are chosen at random from a uniform distribution of those not previously selected in this round. By the end of a full round, it is ensured that all of the leaves that were in the DAG at the beginning of that full round have been reassigned. Batched reassignment is parameterized by the number of rounds and number of leaves to reassign in each sub-round because it is the most natural parameterization independent of the size of the configuration. The alternative, of parameterizing based only on the number of leaf reassignments to perform, is inadequate because that number would have to change as the size of the configuration design changed.

The number of rounds in batched reassignment can be used to trade-off the solution quality and run-time. In our experiments, setting the number of rounds *n* to between three and ten (depending on the complexity of the workload) is sufficient. This is because we found that we gained the most benefit in the first few rounds. As a possible extension, rather than setting the number of rounds directly, DAD could provide support for a user to specify a time for which they are willing to let DAD run, and leave it up to the tool to determine the corresponding number of reassignment rounds.

Lines 18–24 of Algorithm 1 show the single store reassignment phase of the DAD algorithm. In this phase DAD performs small adjustments by moving individual stores around to optimize secondary goals; for instance, in this phase DAD attempts to balance the load across tightly packed devices. A single store, randomly selected from a uniform distribution, is removed from the DAG and reassigned back into the DAG using single store assignment; this reassignment is repeated *m* times, where *m* is calculated from the fraction of the total number of stores specified by the user. Stores are chosen randomly for reassignment to avoid bias towards any particular comparison function.

Both batched reassignment and single store reassignment need to be able to compare the previous and current device DAGs after making a change, and choose the best, based on the user's specified goals for the configuration. This happens in Lines 17 and 24. A *design comparison function* is used to compare two complete designs: each design is first summarized and then the summaries are compared to choose the best design. The way a design is summarized depends on the particular comparison function. For example, for the goal of minimum price, each design is first summarized by calculating the total price of all DAGs in that design and then the design with the lowest price is chosen. Similarly, for the goal of balanced utilization, the variance in the utilization is summarized for each design and the preferred design is the one with less variance. Because the functions compare summaries, we only need to store summaries of the DAGs, rather than maintaining two complete copies of both the previous and current designs at the same time.

3.3 Single store assignment

Single store assignment takes a single store and a node in a device DAG, and tries to add the store to the subtree of component nodes rooted at that node. The algorithm is essentially a depth-first search to find the best possible leaf node to add the store. For a single store, this search visits all of the component nodes in the DAG (illustrated as ovals in Figure 6) and tries adding the store to each of them in turn. It also explores additional alternatives by transforming existing nodes to different configurations, and by adding new nodes into the DAG to determine if it would be best to assign the store to one of those nodes instead.

To compare two alternative assignments of a single store into a DAG, it is sufficient to compare, for each assignment, just the path from the root of the DAG to the leaf component node at which the assignment is being done. A full design comparison is unnecessary during single store assignment, since a store can only be assigned along a single path, leaving the rest of the device DAG unchanged. Thus, whereas design comparison is used during reassignment, *path comparison* is used during single store assignment to more efficiently summarize and compare just two paths. Single store assignment takes as parameters the best path seen so far, for comparison with the assignment tried in the current invocation, and a path comparison function specified by the user. For example, with respect to the DAG in Figure 6, a path comparison function might compare one path starting at the root and ending at LU_3 on FC-60_1 with another path starting at the root and ending at LU_0 on FC-60_2. If the user goal was minimum price and minimum resource utilization, the appropriate path comparison function would be used to first calculate the total price for the competing paths and pick the one with the minimum price; then, if there is a tie, it would be resolved by computing an average over the resource utilizations in each path and picking the path with the lowest utilization.

At each component node in a path, the single store assignment function updates the

Algorithm 2 Algorithm for single store assignment; invoked from Algorithm 1. The algorithm is a depth first search through the DAG, pruning the search (Lines 6,8) when possible. The procedure also attempts to modify the DAG by either adding nodes, or transforming them to explore additional designs (Lines 15–21).

```
 1: $B$: best path seen so far, based on user goals
 2: $C$: current path being evaluated
 3: procedure Assign_Single_Store(store $s$, node $n$, path $B$, user_goals $G$)
 4:     {Evaluate assignment to current node}
 5:     Add store $s$ to node $n$, updating $n$'s attributes
 6:     if any of node $n$'s attributes are overloaded then return false
 7:     Append node $n$'s attributes to path $C$
 8:     if path $C$ from root to node $n$ is worse than best path $B$, w.r.t. user goals $G$ then return false
 9:     if $n$ is a leaf component node of the device DAG then
10:         best path $B$ := current path $C$
11:         return true
12:     {Recurse to sub-nodes of $n$}
13:     for every sub-node $c$ of node $n$ do
14:         Assign_Single_Store($s$, $c$, $B$, $G$)
15:     for every sub-node creation function $F$ defined in node $n$ do
16:         Create a new sub-node $c$ of $n$ using $F$
17:         if creation of $c$ succeeds then Assign_Single_Store($s$, $c$, $B$, $G$)
18:     for every sub-node transformation function $T$ defined in node $n$ do
19:         for every sub-node $c$ of node $n$ do
20:             Transform sub-node $c$ using $T$
21:             if transformation of $c$ succeeds then Assign_Single_Store($s$, $c$, $B$, $G$)
22:     if any of the recursive calls to Assign_Single_Store returned true then
23:         return true
24:         else return false
```

attributes (illustrated as hexagons in Figure 6) associated with the node, to determine the additional demand placed on a resource by the assigned store. Finally, when the search reaches a leaf node, it assigns the store to that leaf. For example, consider how to determine the effect on Controller A of adding a store to the disk array FC-60_1. To take into account the demands of the streams associated with the new store, the function uses the device performance models to adjust Controller A's utilization. The search then continues down the path to the leaf LU_3, for example. There, it adds the capacity of the new store to the value stored in the LU Capacity attribute.

The single store assignment algorithm Assign_Single_Store is shown in Algorithm 2. Given a store and a component node of the design DAG, it initially evaluates the results of adding the store to the current node (Lines 4–11). First, it calculates the impact of adding the store on the value of each attribute associated with the node (Line 5). It checks whether the store "fits" on the current node: if adding the store results in any attribute value exceeding its maximum (Line 6), the assignment of the store to that path fails. For example, if the controller utilization of the FC-60 exceeds the maximum allowed, then the store cannot be assigned to any of the sub-nodes in that path, so the search along that path fails and returns. At Line 7, comparison information held on the current path is extended to include the new attribute information for the current node, so that when the comparison function is invoked in Line 8 (and, after recursion, at sub-nodes) it has all of the necessary information on the state of the path so far. In Line 8, the procedure checks whether or not the current path is already worse than the best path seen so far for this store assignment. If, for example, the path comparison function (based on user goals) prefers low-cost solutions, the best solution so far has zero cost and this path would require the purchasing of more resources, then the search along this path fails and returns at this point. Then, in Lines 9–11, the procedure checks whether the search has reached a leaf node. If so, then the current path is at least as good as the best at every level, so this assignment is identified as the best assignment so far and the search along this path succeeds and returns.

The procedure then recurses through the sub-nodes (Lines 12–21), to explore alternative nodes further down a path. First the search recurses into each of the *existing* sub-nodes of the current node (Lines 13–14). For example, from the root in Figure 6, the search would initially recurse into the FC-60 nodes. The procedure then attempts to assign the store after instantiating *new* sub-nodes (Lines 15–17). For each node addition function, such as the LU creation function of the FC-60 disk array nodes, the algorithm uses the function to create a new node, and then recurses into it. Finally, in Lines 18–21, the procedure tries to assign the store after *transforming* each of the sub-nodes. For each sub-node transformation function, such as a function to choose higher capacity disks for an LU, or to turn a RAID 5 LU into a RAID 1/0 LU, and for each sub-node, the algorithm transforms the sub-node, and then recurses into the transformed sub-node to continue the search.

To simplify the explanation of the search algorithm, we have omitted all steps where design-DAG states are saved and restored. After we describe speculation, our technique used to limit the amount of state saved and to move efficiently between saved states during the search process, we show Algorithm 3 which adds that necessary state saving and restoring operations to Algorithm 2.

3.4 State management with speculation

To support the rapid try/undo pattern exhibited by the design tool, we need the ability to move quickly between multiple versions of the design DAGs. The requirements are that the operation to create a new version of a design DAG, by modifying a node or a subgraph, should be quick, and the space needed for the versions should be optimized. We introduce a new feature, called *speculation*, to the device DAG data structure, which avoids making complete copies of the device DAG on small updates. We call it “speculation” by analogy to the way in which modern CPUs will speculatively evaluate parts of a program and suppress results if they shouldn’t have been calculated.

Speculation is similar to the features of Multi-trees [Furnas and Zacks 1994], which are DAGs with easily identifiable tree substructures. However, unlike multi-trees, which create a new root and incorporate parts of an existing tree by reference, speculation is implemented by creating new DAGs inside the existing device DAG, temporarily replacing the existing nodes. For efficiency, we use a copy-on-write approach, only making copies of the parts of the design DAG that will change, just before those changes take place. Before a node is modified, copies are made of anything in that node that might change. Before a sub-graph is modified, a copy is made of the all nodes in that sub-graph.

Speculation associates a set of stacks with each node in the device DAG. DAD uses two stacks, but allows particular devices to create additional stacks dynamically. These stacks are used by the search algorithm to save and restore intermediate states of the node.

Speculation's copy-on-write strategy ensures that making multiple copies of an unchanged node is efficient. Speculation supports the following three primitive operations and two composite operations on a node's state.

(1) *Recursive_Save_State(node n, stack s)*: push a new copy-on-write copy of the node *n*'s state on its *s* stack, recursively save the state of all sub-nodes to their corresponding *s* stack.

(2) *Recursive_Restore_State(node n, stack s)*: pop a previously saved state of the node *n* off its *s* stack and restore that state, recursively restore the state of all sub-nodes from their corresponding *s* stack.

(3) *Recursive_Discard_State(node n, stack s)*: pop a previously saved state of the node *n* off its *s* stack and discard that state, recursively, from all sub-nodes, pop and discard the topmost state from their *s* stack.

(4) *Recursive_CopyTo_State(node n, stack s)*:
Invoke Recursive_Discard_State(*n*,*s*); Invoke Recursive_Save_State(*n*,*s*)

(5) *Recursive_CopyFrom_State(node n, stack s)*:
Invoke Recursive_Restore_State(*n*,*s*); Invoke Recursive_Save_State(*n*,*s*)

As we mentioned before, the search algorithm uses speculation to store copies of a node's state on the stacks associated with each node. We need to keep track of state information because the design DAG is a global data structure and the search algorithm is recursive. Before the search algorithm makes a change to a node, it uses speculation to save a copy of that node and of the nodes in its sub-tree; the stacks help to maintain multiple copies of a single node during the search.

The search algorithm sets up speculation to use two state stacks on every node: "trial" and "best". The "trial" stacks are used to store the current state of the nodes during a search, while the "best" stacks store the state of the nodes for the best design found so far.

Algorithm 3 is an extended version of Algorithm 2 that shows where speculation operations are called during execution of Assign_Single_Store. Line 4 of Algorithm 3 calls Recursive_Save_State to save the initial state of node *n* and its subnodes on both their "trial" and "best" stacks. This initial "trial" state is the state to which the node reverts if store assignment fails. This initial "best" state is the best path seen so far, which will be used as the basis for comparison with the store assignment alternatives considered during this execution.

The algorithm evaluates the impact of adding the store to the current node (Lines 5-13). If the assignment to the current node fails, then the node reverts to the initial state saved on its "trial" stack (Line 35) and the algorithm terminates. If the assignment to the current node succeeds and this node is a leaf node then the current state of the node is saved to its "best" stack (Line 12). The algorithm then terminates with the initial state of node *n* being replaced by the better state restored from its "best" stack (Line 34).

If the algorithm recurses to evaluate the effects of assigning the store to sub-nodes of *n* (Lines 14-35), then Recursive_CopyTo_State is called before this recursion begins to save the state of node *n* with the store *s* assigned to it onto its "trial" stack first (Line 15). This enables each alternative sub-node assignment to be evaluated from this same starting point, with this same state being copied from the "trial" stack in lines 19, 24 and 30, before the next iteration of the relevant for-loop.

Algorithm 3 This algorithm expands on the original Algorithm 2 for single store assignment with the addition of all the speculation statements that are necessary.

```
1:  $B$: best path seen so far, based on user goals
2:  $C$: current path being evaluated
3:  procedure Assign_Single_Store(store $s$, node $n$, path $B$, user_goals $G$)
4:    Recursive_Save_State($n$,“best”); Recursive_Save_State($n$,“trial”);
5:    {Evaluate assignment to current node}
6:    Add store $s$ to node $n$, updating $n$’s attributes
7:    if any of node $n$’s attributes are overloaded then goto fail
8:    Append node $n$’s attributes to path $C$
9:    if path $C$ from root to node $n$ is worse than best path $B$, w.r.t. user goals $G$ then goto fail
10:   if $n$ is a leaf component node of the device DAG then
11:     best path $B$ := current path $C$
12:     Recursive_CopyTo_State($n$,“best”);
13:     goto success
14:   {Recurse to sub-nodes of $n$}
15:   Recursive_CopyTo_State($n$,“trial”);
16:   for every sub-node $c$ of node $n$ do
17:     Assign_Single_Store($s$, $c$, $B$, $G$)
18:     if found a better assignment then Recursive_CopyTo_State($n$,“best”)
19:     Recursive_CopyFrom_State($n$,“trial”)
20:   for every sub-node creation function $F$ defined in node $n$ do
21:     Create a new sub-node $c$ of $n$ using $F$
22:     if creation of $c$ succeeds then Assign_Single_Store($s$, $c$, $B$, $G$)
23:     if found a better assignment then Recursive_CopyTo_State($n$,“best”)
24:     Recursive_CopyFrom_State($n$,“trial”)
25:   for every sub-node transformation function $T$ defined in node $n$ do
26:     for every sub-node $c$ of node $n$ do
27:       Transform sub-node $c$ using $T$
28:       if transformation of $c$ succeeds then Assign_Single_Store($s$, $c$, $B$, $G$)
29:       if found a better assignment then Recursive_CopyTo_State($n$,“best”)
30:       Recursive_CopyFrom_State($n$,“trial”)
31:   if any of the recursive calls to Assign_Single_Store returned true then
32:     goto success
33:     else goto fail
34:   success: Recursive_Restore_State($n$,“best”); Recursive_Discard_State($n$,“trial”); return true
      (found better assignment)
35:   fail: Recursive_Restore_State($n$,“trial”); Recursive_Discard_State($n$,“best”); return false (did
      not find better assignment)
```

For each sub-node store assignment, if the comparison function determines that the current store assignment is better than the best seen so far, then Recursive_Copyto_State saves the current state of the node and its subnodes on their “best” stacks (Lines 18, 23 and 29). Once all the sub-node store assignment alternatives have been evaluated, node *n* and its subnodes will be set to the state from the top of their “best” stacks if a successful store assignment was found (Line 34) or they will be reset to their initial state from the top of their “trial” stacks otherwise (Line 35).

The main advantage of speculation is that it eliminates the need to track how the design tool got to a particular configuration. This gives speculation a more flexible approach than standard backtracking [Dechter and Frost 2002]. The traditional backtracking approach makes it necessary to record the operations that would undo all the intermediate transitions. Speculation only has to implement the "forward" modifications to the device DAG; an undo operation involves just deleting the corresponding state and reverting to the previous state. Intuitively, speculation allows us to mark various configurations, in the series of configurations explored by the search algorithm, and transition efficiently between any of them. It is comparable to being able to do arbitrary back and forward tracking through the entire set of explored configurations without the cost of tracking the operations that led to the different configurations. A second advantage of speculation is that it makes switching between different designs linear in the number of changed nodes, rather than the number of operations. As the number of nodes is usually much smaller than the number of operations, this is a substantial improvement.

4. EVALUATION

How well DAD's designs satisfy the application is determined by the quality of search heuristic that DAD uses and the accuracy of the underlying device performance models. We assume that the underlying models are sufficiently accurate, as shown by their authors [Uysal et al. 2001; Merchant and Alvarez 2001; Alvarez et al. 2001; Anderson 2001; Anderson et al. 2002], and focus our evaluation on the search heuristic. Because we demonstrate that DAD generates near-optimal designs for multiple different models, we believe that if the models were improved, DAD would still generate near-optimal designs for the improved models too.

We evaluate DAD using two metrics. Our primary metric is the quality of the solution. The quality of the solution is determined by how well the user's goals have been met. For example, given an input workload, a device model and a goal of minimum system price, the lowest priced valid solution has the highest quality. Our second metric is the running time used by the various approaches, with a more qualitative goal of showing that the performance of the tool is "good enough" to be usable in practice. In particular, we show that DAD is fast enough that initial solutions could be used to help administrators explore various "what if I constrain the design in this way" scenarios, and that when they want a near-optimal design it is still generated relatively quickly. We present sufficient detail about the experiments and evaluation for the reader to understand what we have done and we highlight the important results to illustrate what we have achieved. For completeness, we have made available the graphs for all the evaluations in the technical report [Anderson et al. 2001].

Another possible evaluation would be to show that the designs that DAD generates adequately satisfy the application's requirements. We do not perform this evaluation because it has already been demonstrated convincingly by the Hippodrome paper [Anderson et al. 2002], where DAD (referred to as Ergastulum) is used as part of an automated storage management system. Recall from Section 2, that the models have been validated against real workloads. Since DAD uses some of the same models, the designs it generates should support the workloads.

4.1 Overview of evaluated search algorithms

There are many different possible search algorithms for the device configuration problem. We describe these algorithms by casting the problem as a search for an optimal store ordering. To make this claim, we will informally argue that the following statement is true. Given the set of configuration modification functions in DAD, device models for which removing a store will not increase utilization, and a goal that allows us to move stores between LUs provided the models allow the movement, then a design with the same price as any locally minimal design can be created based solely on the store order. In other words, for every locally minimal design, there exists a store order which will generate a design of the same cost using only the initial assignment phase.

Consider the following transformation of a given minimal design. First, place the leaves of the design in an arbitrary order; note that the stores are assigned to leaf nodes. Selecting the stores in turn from the ordered leaves, move stores to leaves earlier in the order so long as the models allow that movement. Once this equal-price, valid design has been generated, sort the stores in the order of the leaves.

We can now argue that if we use the above generated order of stores to do an initial assignment, we get the same design. Because of the assumption about models (that adding a store never decreases the utilization), the first group of stores will re-generate the first leaf of the adjusted design. From our construction, no other store can fit on the first leaf because we have verified that the models do not allow any other store to be moved to this leaf. Hence the first group of stores will be exactly those that are assigned to the first leaf, the next group of stores will re-generate the second leaf of the adjusted design and so on, until the entire adjusted design has been re-generated.

Two conditions which can be used to show the invalidity of the arguments above are a) limiting transformation functions and b) goals that depend upon the exact nature of store placement. For instance, if the search algorithm is unable to switch a leaf between a RAID 1/0 and a RAID 5 LU after creating the leaf then it will be stuck with the type it creates initially. Similarly the construction is invalid if the goal does not allow us to move the stores in the locally minimal design to an equivalent minimal design. This can happen, for example, with rules such as “stores starting with the letter ‘c’ must go on the third LU of a device”. However, we believe that most of the goals used in practice only depend upon the aggregate design, including overall cost of design, for which our argument is valid.

Because of the relationship between minimal designs and store orders, we can, for comparison purposes, describe all of the algorithms as a search for an optimal store ordering.

—**Minerva:** This system [Alvarez et al. 2001] uses a modified Toyoda [Toyoda 1975] search algorithm. Taking into account both capacity and phased utilization, this algorithm calculates the impact of store assignment on resource usage for each (store, LUN) pair. It initially generates store orders that are likely to fill LUNs in order of increasing cost and, within each LUN, to minimize resource contention. Assignments are then optimized to balance capacity and utilization on the LUNs.

—**Integer-linear programming:** This approach [Nemhauser and Wolsey 1988] uses branch and bound to perform an optimized exhaustive search. In a branch-and-bound tree, every subtree can be viewed as an attempt to pack a store, and thus the whole search can be seen as yet another way of generating store orders.

—**Repeated initial assignment:** This method generates some number of uncorrelated, random store orders, and then picks the one that yields the best design.

—**DAD:** This system performs initial assignment and then some rounds of reassignment. Initial assignment generates an initial store ordering, and reassignment attempts to take a subset of the stores and splice them in new locations in the ordering to get a better design.

Using reassignment, DAD usually performs better than repeated initial assignment. The intuition behind this can be understood by considering the following analogy. Assume we have ten bins, each containing a number between 0 and 1. Our goal is to minimize the sum across all the bins. As an equivalent to reassignment or repeated initial assignment, we use a randomization procedure to pick a new number for some set of the bins. If we repeatedly apply our randomization procedure to ten random bins, and accept the results if the sum has been improved by their new numbers, then over time we will slowly find better and better sums, in the same way that repeated initial assignment (or "reassignment" across all the bins) slowly improves the design. However, if we apply the randomization procedure to each bin individually, and accept the new number in the bin if the sum has been improved, then the sum decreases much faster. This explains the benefit of reassigning a single LU. Additionally, although this analogy doesn't extend to the following case, where there is sufficient correlation between the choices then more benefit may be gained by adjusting multiple bins simultaneously. This explains why reassigning multiple LUs at the same time can work better than reassigning a single LU.

4.2 Solution quality for real models

Determining the quality of solution generated by a heuristic for an NP-hard problem is very difficult. For simple models, or for simple workloads, quality bounds can be determined [Kenyon 1996; Fernandez and Lueker 1981; Johnson et al. 1974b]. We, however, are interested in complex models and workloads because they represent the devices and applications found in real systems. We therefore have to use alternative techniques to evaluate the quality of our solutions.

We estimate the quality of solutions by first sampling a large collection of designs generated using only initial assignment and a random store order. In Section 4.1, we explained that an initial assignment can be associated with a local minima given appropriate store ordering. By calculating a large number of designs generated from random store orders, we lower the probability that a substantially better design exists. In the next section, we evaluate whether reassignment can "fix" poor initial assignments more quickly and effectively.

Sampling a large number of designs generated from random store orders also provides insight into the difficulty of finding good solutions to a particular workload/model combination. While the overall problem of finding designs is NP-hard, a particular problem may be very simple. For example, the problem of packing a collection of stores that are all the same size and perform no I/Os into fixed size devices of a single type can be trivially solved in fixed time because all that needs to be determined is how many stores fit in one device, and then the required number of devices can be fully packed, with potentially a few left-over stores in one last device. Such a solution would also be optimal, if load balancing was not also important. While none of the workload/model combinations that we test are that simple, in practice some of them only generate a single system cost across many tests, leading us to believe that a system with that cost is probably optimal, and that those combinations are trivial to solve.

We performed the experiments using both FC-30 [Hewlett-Packard Company 1998] and FC-60 [Hewlett-Packard Company 2000] device models, using nine different configuration options and 16 different workloads for a total of 144 workload-option pairs. The workloads

Table I. Configurations explored in the initial assignment experiments. The sizes of the RAID groups were chosen based on the natural sizes for the array. For example, the FC-30 has 5 SCSI buses for the disks, leading to a balancing of the 5 disk RAID 5 LU across all the buses. Because RAID 1/0 can only be used with an even number of disks, a 4 disk LU is the largest that can survive a SCSI bus failure. The FC-60 has 6 SCSI buses, so that size is used for both RAID 1/0 and RAID 5. Restricting the FC-60 to use only 9 GB disks gives mostly uninteresting results, which can be found in [Anderson et al. 2001].

array	*LU type*	*disk size (GB)*	*name*
FC-30	4 disk RAID 1/0 5 disk RAID 5 4 disk RAID 1/0 and 5 disk RAID 5	4	-FC-30-r1 -FC-30-r5 -FC-30
FC-60	6 disk RAID 1/0 6 disk RAID 5 6 disk RAID 1/0 and 6 disk RAID 5	9, 18, 36, 72	-FC-60-r1 -FC-60-r5 -FC-60

used for these experiments are all summaries of traces of real workloads and traces of application benchmarks. To generate our large collection of designs, we have performed at least 500 runs of DAD for each workload-option pair, and at least 100 runs for each unique system price found for that workload-option pair. In total, we have performed 1,169,816 individual runs for this evaluation.

As described above, DAD finds optimal or near-optimal designs for nearly all of the workload-option pairs. For 87 of the 144 workload-option pairs (about 60%), the difference between the least expensive and most expensive system price was less than 5%, indicating that DAD was able to find optimal or at least near-optimal designs in all these cases. In fact, for 46 of these 87 workload-option pairs, we found that all the designs for a pair had the same price, indicating that DAD probably generated the optimal solution. In general, larger workloads and more flexible design rules lead to greater variability in system price. To demonstrate how DAD handles the more complex real-world workloads, we focus on the results for 10 of the workload-option pairs, illustrated in figure 7, that proved to be the more interesting, challenging problems for DAD. The configuration options for these 10 experiments are described in Table I. The seven workloads used are described below. These workloads are summaries of traces of real workloads and traces of application benchmarks.

—*openmail*. The OpenMail trace summarizes the real workload of an email server in one of HP's data centers with about 2000 active users accessing 98 stores in the space of one hour.

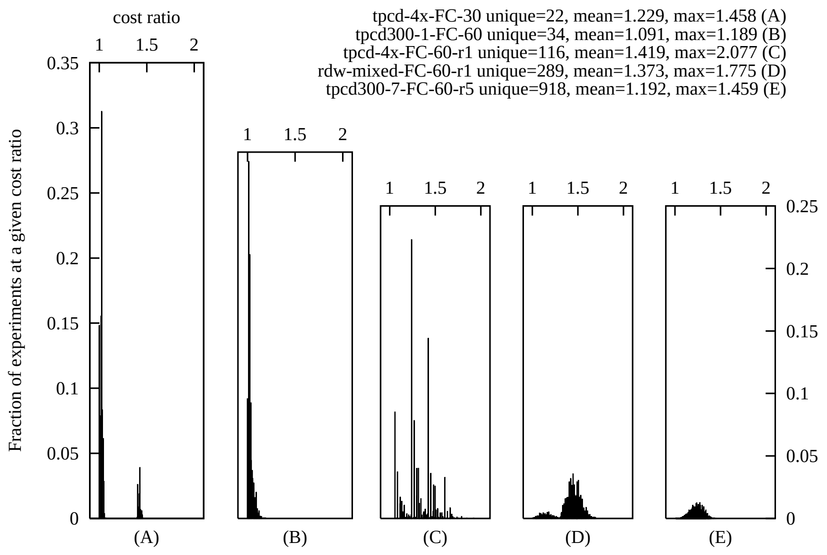
—*rdw-*{*intense, mixed*}. A 1 TB retail data warehousing workload with 325 stores. This realistic simulation of the decision support operations used to analyze supermarket basket information exhibits the complex I/O behavior of a real-world application. The *intense* variant has 25 intense (15+ minutes individually) simultaneous queries. The *mixed* variant has 50 quick (few seconds individually), and 25 easy (tens of seconds individually) simultaneous queries.

—*tpcc-midrange*. A midrange TPC-C configuration with 156 stores where the database performs about 16.5K tpmC.

—*tpcd-4x*. Four copies (to make it large enough to be interesting) of a 10 GB TPC-D-like decision-support benchmark running queries in parallel, with uncorrelated overlaps between the four copies. The input had 1266 stores.

—*tpcd300-*{*1,7*} Two of the most I/O intensive queries, one and seven, in a 300GB-scale TPC-D benchmark. The input had 923 stores.

Despite the fact that, as the graphs in figure 7 show, the storage system design problem



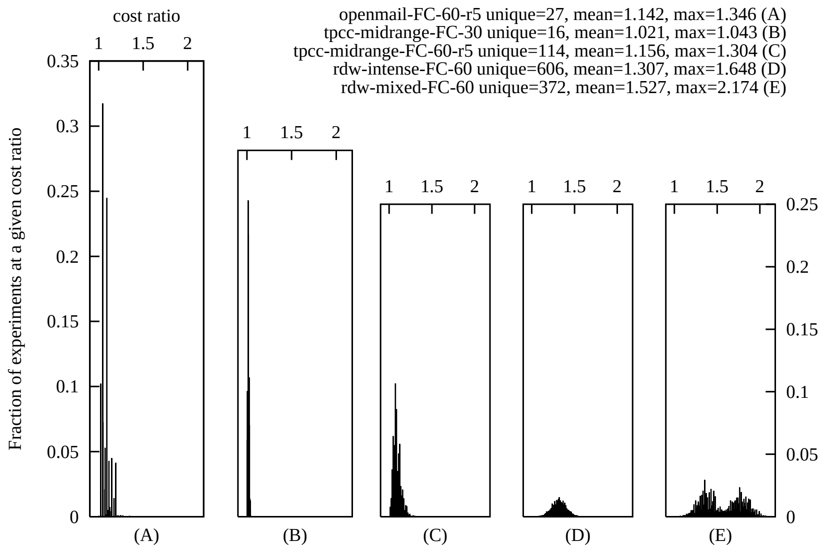


Fig. 7. Subset of results from running 1,169,816 random experiments with different workloads and device parameters. The results show that the storage design problem is complex for these workloads and that just using initial assignment would be a poor choice. These graphs show results from runs of ten of the most interesting experiments, five (A-E) in each graph. The results of one experiment are shown as the range of system prices relative to the least price found during runs of that experiment. Each column represents the fraction of runs found to have a given price; from the left-most column (at the center of the x-axis label for that experiment) showing the fraction of runs at the lowest price to the right-most column for that experiment showing the fraction of runs at the highest price found. The number of different prices found, mean relative price, and the maximum relative price are shown in the captions. Consider experiment (A) from the top graph: this shows 22 unique prices in two groups, a large left one and a smaller right one. Experiment (B) shows only one group, and (C) shows that same structure replicated four times. (D) and (E) in each graph show single and double hump normal-like distributions, where (D) on the left is heavily weighted toward the right.

can be very complex, DAD does produce a valid design for every experiment, even though some of the experiments generate hundreds of distinct system prices. It is easier for DAD to produce a design that meets the minimal system price goal for some experiments, such as *tpcd300-1-FC-60*, because the results in these experiments are heavily weighted toward the minimal price configurations. It is more difficult to produce good designs for other experiments, such as *rdw-r10-mixed-FC-60*, where the results are weighted towards more expensive configurations. As illustrated in these graphs, many of the runs generate normal or bimodal distributions of system price.

The experiments show that the best results are very rare, implying that using only initial assignment would be a poor choice. For 56 of the 144 workload-option pair experiments, less than 10% of the runs generated the best-seen result. For 20 of the experiments, less than 0.01% of the runs generated the best-seen result.

The experiments also show that mistakes in initial assignment can be expensive. For the *rdw-r10-mixed-FC-60* experiment, the ratio between the most expensive and the least expensive configuration found is about 2.1. For 47 of the 144 experiments, the ratio between the most expensive and the least expensive configuration is over 1.2.

Thus we can see that initial assignment produces valid solutions, but is not likely to produce near-optimal solutions. However, these experiments do show that the runtime of initial assignment is very good. The simple designs, resulting in roughly $150,000 storage systems take only 2 seconds on a 1GhZ Pentium III; the most complex runs take about 11 minutes to calculate a roughly $750,000 storage system. These runtimes are fast enough that initial assignment could be used to generate designs for “what if” analysis by a system administrator, especially since, if they constrain part of the problem, the runtime will decrease.

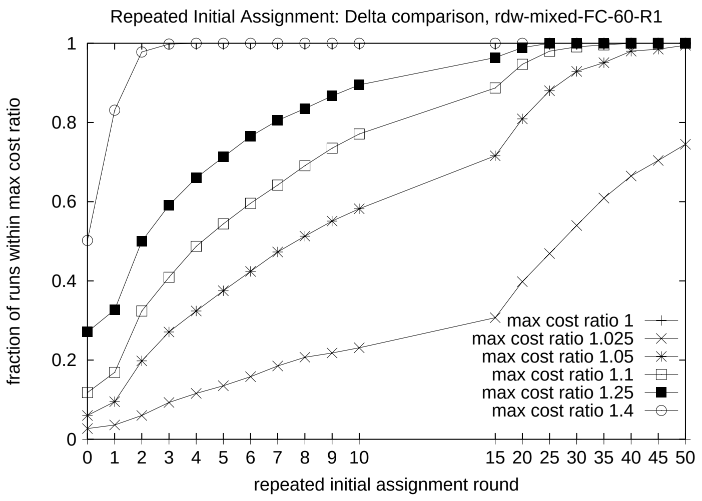
The wide distribution of results that we have found illustrates that real-world workload and device combinations can lead to complex design problems, and that initial assignment is insufficient to handle them. Thus, we need reassignment to help us address these complex problems.

4.3 Solution quality with reassignment

One of the key ideas in DAD is the use of reassignment to try to improve an initial design. To test its effectiveness, we compared reassignment against repeated initial assignment. Section 4.1 explained why reassignment can be expected to do better; and this is exactly what we found, even for the most challenging workload-option pair that we tested.

For all but one of our experiments, reassignment finds nearly optimal results (within 5% of the best result) for almost all of the runs (above 90%) within 2-5 rounds. Figures 8(a,b) compare repeated initial assignment (a) with reassignment (b) for the experiment on which reassignment had the most difficulty in generating the best result we saw in any run. As shown, repeated initial assignment is better at removing the worst case, given sufficient rounds of repeated initial assignment. However, even for this very difficult experiment, we can see that reassignment is doing well: 60% of the runs are within 2.5% of the best seen cost within 10 rounds whereas repeated initial assignment is not doing quite as well.

Recall that, in Algorithm 1, there are several options that can be set for reassignment: the comparison function used (based on the user’s goals), the number of rounds of batched reassignment n of all the stores on r LUs in the design DAG and the number of rounds of single store reassignment m. We demonstrate which options produce the best results, by showing the highlights of about 200 different options for reassignment. Surprisingly,



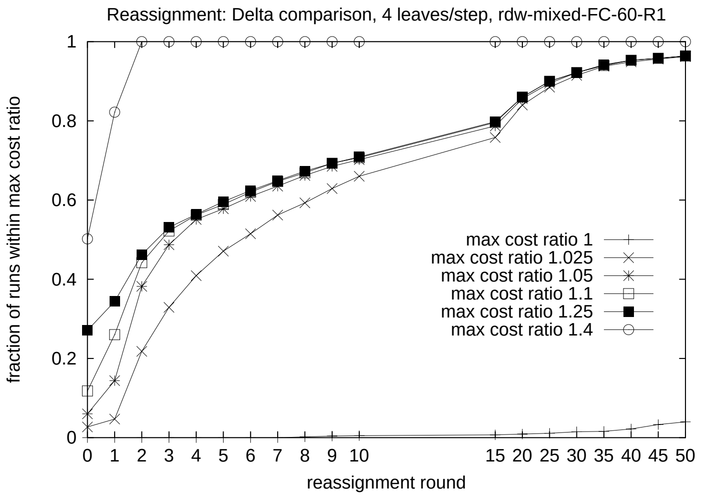


Fig. 8. Comparing repeated initial assignment with reassignment for our most difficult workload-option pair. All our other results show that reassignment is substantially better than repeated initial assignment. The results here are mixed, but show how well reassignment does even with this difficult experiment: a larger fraction of runs using reassignment produce optimal (max cost ratio 1) or near-optimal (max cost ratio 1.025) results within 50 rounds, compared with those using repeated initial assignment. The x-axis shows the number of rounds done within a run, for repeated initial assignment in the top graph and for reassignment in the bottom graph. Results accumulate over rounds. The y-axis shows what fraction of runs are within the specified maximum cost ratio after each round. The Delta comparison function for the top graph compares the current round of initial assignment of all stores with the best round seen so far. The Delta comparison function for the bottom graph compares the current round of reassignment of stores from four LUs with the best round seen so far. Repeated initial assignment does better for the less than optimal 1.25 cost ratio, where it gets the results of all the runs within 25% of the best cost after 25 rounds, compared with reassignment which does not achieve that result within 50. Reassignment is substantially better than repeated initial assignment for the 1.025 cost ratio, getting results for 60% of the runs within 2.5% of the best cost after only 10 rounds, whereas runs using repeated initial assignment takes 45 rounds. A growing fraction of runs have results that match the best result ever seen for reassignment (the line on the bottom graph closest to the x-axis), but no runs using repeated initial assignment find that result at all (the line for max cost ratio 1 on the top graph runs along the x-axis).

as we explain below, we find that a comparison function that prefers assignments with the maximum utilization produces better quality solutions.

To test reassignment, we start with two of the designs for each distinct cost from the initial assignment experiments described in Section 4.2. We then run 50 rounds of reassignment (*n*=50), using five different reassignment comparison functions, and five different counts (*r*=1–5) for the minimum number of LUs to reassign. Using different comparison functions demonstrates the effect of changing the criteria shown in Line 8 of Algorithm 2, which returns false if the current path is worse than the best path. The effect of varying *r*, the number of LUs to reassign, is demonstrated in Lines 10–14 of Algorithm 1 where all stores are deleted from *r* leaves. We try reassigning between 1 and 5 LUs to test out a range of reassignment functions. We also try reassigning all of the LUs to simulate repeated initial assignment. For the results presented here, we use a minimal price design comparison function. For the path comparison function, we first select the least expensive assignment and, if two assignments have the same cost, then:

—*MinAvgAll*: Favor the assignment with the lower mean utilization of all attributes, if the difference is at least 5%. If the assignments are the same to within 5%, consider the choices as equivalent (and discard the current store assignment, since it does not improve on the best one seen so far).

—*MaxAvgAll*: Favor the assignment with the higher mean utilization of all attributes, if the difference is at least 5%. If the assignments are the same to within 5%, consider the choices as equivalent.

—*Delta*: Favor the assignment with the lower increase in the disk utilization attribute. If the increases are within 0.1% of each other, select as per *MaxAvgAll*.

The intuition behind the *MinAvgAll* function is that, given two equal-cost solutions, the one that uses the least resources is probably the best. Conversely, the intuition for *MaxAvgAll* is that between two equal-cost solutions, the one that is packing things the tightest is probably the best. For *Delta*, the intuition is first to select the result that increases the utilization the least, and then try to pack tightly. The selections of 5% and 0.1% are arbitrary, and are there to consider solutions that are about the same as equivalent. A much smaller value was used for Delta because increases in utilization are typically much smaller than the total utilization.

It is up to the comparison function to choose how to handle solutions which are considered equivalent. Recall that Line 8 of Algorithm 2 says "**if** path from root to node *n* is worse than best path B, w.r.t. user goals, **then return** false". A comparison function may define worse as strictly worse, in which case the search algorithm will ultimately choose the last of the equivalently good choices. Alternatively, if worse is defined as strictly worse or equal, then the search algorithm will choose the first of the equivalently good choices. Since the choices are equivalent anyway, it doesn't matter which approach is actually taken.

Figure 9 shows that *MaxAvgAll* performs substantially better than *MinAvgAll*, a result that is consistent across all of our experiments. The explanation for this is as follows. All the stores in an LU must be moved elsewhere, before an LU can be removed from a design. To move stores elsewhere, the solver needs to generate tighter and tighter packings of stores on other LUs in the design. However, the chance of removing an LU while using the *MinAvgAll* comparison function is reduced because this function prefers to place stores on the least-tightly-packed LU, hence reducing the chance of removing an LU.

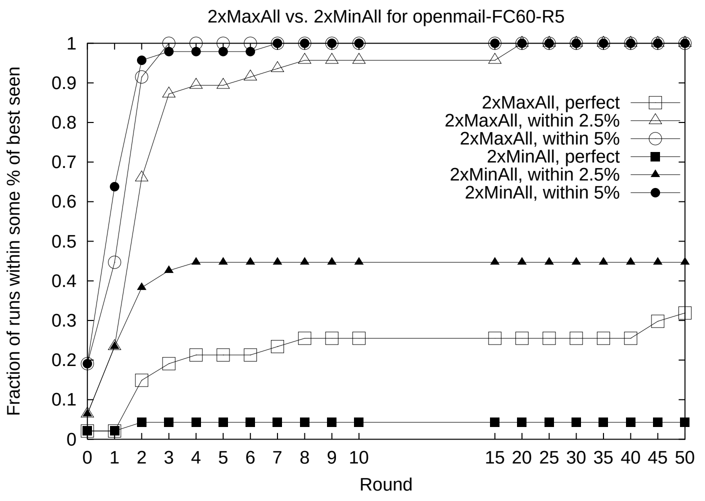


Fig. 9. Evaluating comparison functions for reassignment, MaxAvgAll does significantly better than MinAvgAll. For a representative experiment openmail-FC60-R5, we take initial assignments for each distinct system cost and run 50 rounds of reassignment for each of the two comparison functions. We show, after each of the 50 rounds, what fraction of the results have now come within 5% or 2.5% of the best result seen, or have reached the best result seen. The 2x prefix indicates that in each of these cases, all the stores from 2 LUs were reassigned in a round. For example, the perfect curve for MaxAvgAll, shows that after 10 rounds, 25% of the runs have achieved the best result ever seen, whereas for MinAvgAll only 5% of the runs have done as well. Similarly, MinAvgAll never gets more than 50% of the runs within 2.5% of the best seen, whereas MaxAvgAll gets all the results within 2.5% after 20 rounds.

4.4 Comparison with Minerva

Having completed our direct evaluation of DAD, we now compare it to Minerva [Alvarez et al. 2001], the only other tool we are aware of that tries to design disk array configurations. Minerva uses a similar input specification to DAD, and we have adapted DAD to handle the inputs to Minerva, as well as implementing the same performance models in DAD that are used by Minerva.

To review, Minerva first selects a RAID level for all of the stores in the input workload. Then it runs a simple solver which determines the number and configuration of a set of disk arrays. Next it runs a complex solver which assigns the stores onto the disk arrays, subject to the model constraints and the selected RAID levels. If all stores are not successfully assigned, additional arrays are added and the remaining stores are assigned onto those arrays. To minimize cost, Minerva then repeats the assignment step onto the set of devices generated by the previous loop and prunes out unused devices. Finally, it performs a reassignment with the goal of balancing load; if that reassignment is successful, it is the output, otherwise the previous successful assignment is the output.

Architecturally, DAD improves over Minerva by combining a number of steps that Minerva keeps separate. By separating the steps of selecting RAID levels and device allocation/configuration from the assignment step, Minerva artificially restricts the possible designs, sometimes leading to poorer quality solutions. Furthermore, the complex solver makes inefficient use of the models, which makes Minerva runtimes significantly worse than DAD runtimes.

Table II. A comparison of solution quality between Minerva and DAD, for the “pseudo-realistic” workloads used in the Minerva evaluation [Alvarez et al. 2001]. The max/min price is the ratio between the maximum and the minimum price calculated using the multiple random initial assignment technique. The max and min price were calculated using the Minerva array configuration rules, which restrict the array configuration to 4 disk RAID 1/0 LUs and 4 disk RAID 5 LUs. The ratio shows that the Minerva workloads are not very challenging, in comparison with those used to evaluate DAD’s solution quality, as described in Section 4.2. Furthermore, for 4 of the 5 workloads a random best fit solver has >85% chance of finding the optimal solution. For the dss-p workload, the $136460 solution was found in 1 of 515 random runs. DAD has been run with two sets of array configuration rules: the first column shows results using the same array configuration as Minerva, while the second shows results when the array configuration is less restrictive. In each case, the price and improvement ratio, compared to the Minerva price, are given.

	max/min	*Minerva*	*DAD 4-R1/0; 4-R5*		*DAD 2,4 R1/0; 3-5 R5*	
workload	*price ratio*	*price ($)*	*price ($)*	*ratio*	*price ($)*	*ratio*
filesystem	1.01	422 147	332 504	1.27	330 642	1.28
scientific	1.00	310 160	310 160	1	302 712	1.02
oltp	1.01	748 134	668 732	1.12	667 801	1.12
dss-p	1.05	140 184	140 184	1	136 460	1.03
fs-light	1.00	169 976	169 976	1	166 252	1.02

Table III. A comparison of run-time (in seconds) between Minerva and DAD. The times shown for both Minerva and DAD are for generation of designs using Minerva’s restrictive 4 disk RAID 1/0 and 4 disk RAID 5 array design rules. Minerva’s final design runtimes have a range because, while the Minerva evaluation includes the time to re-evaluate the configuration to check the device utilization, that time should not be necessary to generate a design. Both DAD’s runtime and (in brackets) the relative speedup in comparison with the corresponding Minerva runtime are given for DAD’s first and final designs.

	Minerva first	*Minerva final*	*DAD first design*		*DAD final design*	
workload	*design runtime*	*design runtime*	*runtime*	*(speedup)*	*runtime*	*(speedup)*
filesystem	53	111-227	1.30	(40.8x)	11.18	(9.9-20.3x)
scientific	55	110-277	0.84	(65.5x)	6.18	(17.8-44.82x)
oltp	189	298-492	2.49	(75.9x)	27.41	(10.9-17.9x)
dss-p	223	453-1429	7.54	(29.6x)	43.19	(10.5-33.1x)
fs-light	58	151-371	1.09	(53.2x)	7.71	(19.6-48.1x)

To do our comparison directly with Minerva, we use the “pseudo-realistic” workloads and configuration rules from the Minerva paper [Alvarez et al. 2001]. Minerva uses a set of optimization strategies and heuristics that includes a branch and bound strategy for array allocation and best fit heuristics for store assignment. Table II shows that, for some of the workloads, DAD finds solutions that are between 2-28% better than Minerva and, in fact, just using the initial assignment stage, DAD finds solutions within 5% of the best solution for all the workloads. Table III shows that DAD is 30–76 times faster than Minerva at generating an initial assignment, and 10–20 times faster at generating the final assignment. Minerva’s long runtimes limit its usefulness to administrators who want to explore multiple design constraints or goals interactively or iteratively.

A comparison of the tagging approach used in Minerva and the adaptive approach used in DAD [Anderson et al. 2002] shows that tagging is a very poor approach. Table IV shows that tagging increases the price by 6.3%–14.3% on average, and 25%–50% in the worst cases of those tested. It also shows that the more complex FC-60 disk array presents more difficulties for tagging. Part of the difficulty comes from the assumption in tagging that a particular RAID level has a fixed capacity. While this is true for the FC-30 experiments, which use fixed sized disks and a fixed number of disks at each RAID level, it is not true of the FC-60 experiments, which use multiple disk sizes with a fixed number of disks at each RAID level.

Table IV. A comparison of the better tagging approaches summarized from [Anderson et al. 2002]. 18 different tagging approaches were tested. For the FC-60, there was one tagger which was the best in the tested set; for the FC-30, the results for two taggers are shown, since it was less clear which was the best. The price over-run was the percentage increase for the tagged design above the best design for a particular workload and array. The mean was taken across 12 different workloads.

array	*tagger*	*mean price overrun*	*max price overrun*
FC-30	Best-Tagger-1	7.2%	25%
FC-30	Best-Tagger-2	6.3%	44%
FC-60	Best-Tagger	14.3%	50%

The simple allocator used by Minerva substantially limits Minerva's ability to handle disk arrays effectively. The allocator is essentially a simple, poor solver that only calculates how many arrays are needed and chooses settings for their configuration options, but it doesn't assign stores. Because it has no information on the placement of stores or interaction between the stores, it can make poor choices, which later limit the assignment step. For the Minerva evaluation, which only uses one size of disk and a limited range of configurations, this is only a slight problem. However, a mid-range disk array such as the FC-60 has many different sizes of disks and a wider variety of configurations, and high-end disk arrays offer the option of trading performance for capacity on individual drives. The allocator would be unable to handle this complexity without becoming a full-fledged solver, which is precisely the approach used in DAD, where the allocation and assignment steps are combined.

Finally, Minerva's solver uses the Toyoda algorithm [Toyoda 1975] which makes inefficient use of the disk array performance models by re-calculating the interaction between all store assignments and devices at every step of the algorithm. This prohibits the Toyoda algorithm from using incremental models. In comparison, the DAD implementation makes efficient use of the disk array performance models, by taking full advantage of their incremental models, which does result in improved runtimes. Furthermore, the Minerva solver is forced to consider store assignment to every LU in turn, because it treats each LU as a separate disk. By comparison, DAD can avoid trying to assign a store to each of an array's LUs when the array is already limited by a fully-utilized controller, because it uses the device DAG to represent components such as array controllers higher up in the DAG than LUs, enabling it to prune parts of the search space during store assignment.

4.5 Comparison with integer-linear programming

Finally, we compare DAD's results with solutions that are provably optimal. The optimal results are generated by a design tool that uses integer-linear programming (ILP) [Nemhauser and Wolsey 1988] and a branch-and-bound search to perform an optimized exhaustive search. Unfortunately, there is no known way to cast the complex models that accurately describe disk arrays into the ILP formulation. Therefore, because we have to use simplified, highly inaccurate ($>$1000%) performance models, these comparisons can help us understand the quality of DAD's solutions, but the ILP design tool cannot be used in practice.

We first performed comparisons with random workloads that had no performance requirements, only capacity requirements. The capacity requirements were generated using a variety of distributions. DAD found optimal solutions 80%–90% of the time, and was always within 2.5% of the optimal solution.

Second, we performed experiments with an adversarial workload that is very difficult for randomized algorithms to handle because, compared to the total number of possibilities, the fraction of optimal solutions is vanishingly small. For these experiments, DAD's solutions are always within 5% of optimal.

Third, we added in some simplified utilization models, and tested with both random workloads and with workloads based on realistic models. For these workloads we found that DAD and the ILP design tool found the same solutions.

These results emphasize the additional complexity of handling disk array utilization with realistic workloads. We found one case where the solver could perform over 1.3 times worse than the best result seen, but in these experiments without realistic performance models, the worst case was only 1.05 times worse.

Even if the complex disk-array models could be merged into the ILP design tool, the runtime of the ILP approach is prohibitive, as it took 50–1000 times longer than DAD. For capacity-only experiments, with 500 stores, DAD took 1–2 minutes to calculate a solution, whereas the ILP solver took 0.5–3.5 hours. Adding in the simple performance models increased the runtime such that an ILP run with 100 stores took about 3 hours, and a run with 200 stores took about two days, while DAD found both optimal solutions in less than a minute.

5. RELATED WORK

Buzen [Buzen et al. 1978] proposed a capacity planning tool in which workloads have types; possible types are time sharing, transaction processing and batch processing. The requirements of each type are assumed to be the same. Devices are modeled by grouping devices into groups where only one of the groups can service a request at once. DAD is a more versatile tool, free of simplifying assumptions such as all workloads having the same requirements and device grouping.

Hill [Hill 1994] developed and patented a scheme for the assignment of a workload to devices, based on attributes of I/O rate and capacity. It is not clear how well Hill's scheme would handle more complex attributes that are captured in our workload descriptions and used in the device performance models.

Shriver [Shriver 1996] proposed a formalization of the storage design problem that models workload units with various attributes, objective functions for specifying goals and constraints on devices that can contain workload units. Forum [Borowsky et al. 1997] implemented that formalization and handled storage design for independent disk systems as a multi-dimensional constrained bin-packing optimization problem using an adaptation of the complex Toyoda [Toyoda 1975] bin-packing heuristic. Minerva [Alvarez et al. 2001] extended Forum to handle disk arrays. DAD is an outgrowth of all that work, benefiting from the previous work on performance models, workload specification, etc., but using new search heuristics and a new representation of storage configuration designs that enable it to run faster and generate as good or better designs than other approaches.

Existing solutions to the file assignment problem [Dowdy and Foster 1982; Wolf 1989] use heuristic optimization models to assign files to disks to get improvements in I/O response times. The file allocation schemes described in [Weikum et al. 1991; Scheuermann et al. 1998] automatically determine an optimal stripe width for files, and stripe those files over a set of homogeneous disks. They then balance the load on those files based on a form of "hotspot" analysis, and swapping file blocks between "hot" and "cold" disks. DAD can select the appropriate number of devices to use, supports RAID systems, and uses more sophisticated performance models to predict the effect of system modifications.

The AutoAdmin index selection tool [Chaudhuri and Narasayya 1998] can automatically "design" a suitable set of indexes, given an input workload of SQL queries. It has a component that intelligently searches the space of possible indexes, similar to DAD, and an evaluation component (model, in DAD terms) to determine the effectiveness of a particular selection based on the estimates from the query optimizer.

Muse [Chase et al. 2001] controls server allocation and energy-conscious, adaptive resource provisioning for Internet hosting centers. Unlike DAD, it focuses on allocating computational resources. Its resource allocation framework is based on an economic model that factors in the trade-offs between the service quality and the cost.

The Appia [Ward et al. 2002] system automatically generates network fabric designs. It is designed to take the storage configuration design output from DAD and add a network topology design to it. Network design provides an interestingly different design problem compared to storage design. Because Appia's search space is larger and more challenging than DAD's, with less natural structure to guide the search, it uses substantially different heuristics. Its performance models are simpler than those used by DAD and its assignment space is a little simpler. Appia already uses some randomization in its approach, but adding randomized reassignment might be beneficial, as it has been shown to help DAD substantially.

6. CONCLUSION

In this paper, we have presented the Disk Array Designer (DAD) tool, a new and efficient way of solving the storage configuration design problem. We have presented the algorithm used by DAD to explain how it generates designs. We have established that it produces near-optimal solutions and is substantially faster than other approaches. In summary, we believe that DAD is a good way to quickly solve the storage configuration design problem.

Possible extensions to DAD include application of its approach outside of the storage system domain; in particular, we believe that we could configure and map application host requirements onto appropriate hosts in the N-tier architecture, so that it could handle both the storage and host configuration in a data center. Because it depends on the structure of the "devices" to guide the search, we do not believe that its techniques would work efficiently for selecting a network topology, or other problems with highly flexible device structure. We could verify that DAD works efficiently for choosing between multiple array types, but we expect that in practice sites will have a existing, heterogeneous set of devices, and will be interested in new configurations that only purchase a single type of device, so this functionality may not be necessary. Finally we are considering extensions to DAD that allow us to handle multiple data centers, and may evaluate the effect of constraints that require arrays with availability at or above some level.

7. ACKNOWLEDGMENTS

We would like to thank Daniel Kan for adding support for the FC-60 to DAD, and Liz Borowsky, Jeff Chase, Greg Ganger, Kimberly Keeton, Emre Kiciman, Yasushi Saito, Liddy Shriver, Alistair Veitch, and John Wilkes for their feedback on the paper. We would also like to thank the anonymous referees for their valuable comments which indeed helped us improve the presentation of the paper considerably.

REFERENCES

ALLEN, N. 2001. Don't waste your storage dollars: what you need to know. Research note COM-13-1217, Gartner Group. March. http://www.gartner.com.

ALVAREZ, G., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. 2001. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems 19,* 4 (Nov.), 483–518.

ANDERSON, E. 2001. Simple table-based modeling of storage devices. Technical note, HPL–SSP–2001–4, HP Labs. July. http://www.hpl.hp.com/SSP/papers/.

ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. 2002. Hippodrome: running rings around storage administration. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Monterey, CA, 175–188.

ANDERSON, E., KALLAHALLA, M., SPENCE, S., SWAMINATHAN, R., AND WANG, Q. 2001. Ergastulum: an approach to solving the workload and device configuration problem. http://www.hpl.hp.com/SSP/papers/.

ANDERSON, E., SWAMINATHAN, R., VEITCH, A., ALVAREZ, G., AND WILKES, J. 2002. Selecting RAID levels for disk arrays. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Monterey, CA.

BOROWSKY, E., GOLDING, R., MERCHANT, A., SCHRIER, L., SHRIVER, E., SPASOJEVIC, M., AND WILKES, J. 1997. Using attribute-managed storage to achieve QoS. In *Proceedings of the 5th International Workshop on Quality of Service*. Kluwer, Columbia University, New York, USA, 199–202.

BUZEN, J. P., GOLDBERG, R. P., LANGER, A. M., LENTZ, E., SCHWENK, H. S., SHEETZ, D. A., AND SHUM, A. 1978. BEST/1—design of a tool for computer system capacity planning. In *AFIPS National Computer Conference (NCC), S. P. Ghosh and L. Y. Liu, Ed.* AFIPS, Montvale, N.J., 447–455.

CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., AND DOYLE, R. 2001. Managing energy and server resources in hosting centers. In *18th ACM Symposium on Operating System Principles (SOSP'01)*. ACM Press, Chateau Lake Louise, Banff, Canada, 103–116.

CHAUDHURI, S. AND NARASAYYA, V. 1998. AutoAdmin "What-if" index analysis utility. In *SIGMOD International Conference on Management of Data*. ACM Press, Seattle, USA, 367–378.

COFFMAN, JR., E. G., GAREY, M. R., AND JOHNSON, D. S. 1997. Approximation algorithms for bin packing: a survey. 46–93.

DECHTER, R. AND FROST, D. 2002. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence 136,* 2 (apr), 147–188.

DOWDY, L. W. AND FOSTER, D. V. 1982. Comparative models of the file assignment problem. *ACM Computing Surveys 14,* 2 (June), 287–313.

FERNANDEZ, W. AND LUEKER, G. 1981. Bin packing can be solved within 1+ε in linear time. *Combinatorica 1,* 4, 349–55.

FURNAS, G. W. AND ZACKS, J. 1994. Multitrees: enriching and reusing hierarchical structure. In *Proceedings of the Human Factors in Computing Systems CHI '94 Conference*. Boston, MA, 330–336.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability*. W.H. Freeman and Company.

Hewlett-Packard Company 1998. *Model 30/FC High Availability Disk Array – User's Guide*. Hewlett-Packard Company. Pub. No. A3661-90001.

Hewlett-Packard Company 2000. *HP SureStore E Disk Array FC60 - Advanced User's Guide*. Hewlett-Packard Company.

Hewlett-Packard Company 2001. *HP Surestore Disk Array XP256 - Configuration Guide*. Hewlett-Packard Company. Chapter 4.7.5 HP e3000 Business Servers Configuration Guide.

HEWLETT-PACKARD DEVELOPMENT COMPANY. 2003. HP Managed Services: delivering more storage value on demand. HP StorageWorks case study.

HILL, R. A. 1994. System for managing data storage based on vector-summed size-frequency vectors for data sets, devices, and residual storage on devices. US Patent 5,345,584, 6 September 1994.

JOHNSON, D., DEMERS, A., ULLMAN, J., GAREY, M., AND GRAHAM, R. 1974a. Worst case bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing 3*, 299–325.

JOHNSON, D. S., DEMERS, A., ULLMAN, J. D., GAREY, M. R., AND GRAHAM, R. L. 1974b. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing 3,* 4 (Dec.), 299–325.

KENYON, C. 1996. Best-fit bin-packing with random order. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*. ACM, Atlanta, Georgia.

KENYON, C. 1997. Best-fit bin packing with random order. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*.

LAMB, E. 2001. Hardware spending matters. *Red Herring*, 32–33.

LOAIZA, J. 2002. Optimal storage configuration made easy. White paper 295, Oracle Corporation.

MERCHANT, A. AND ALVAREZ, G. A. 2001. Disk array models in Minerva. Tech. Rep. HPL–2001–118, HP Labs. Apr. http://www.hpl.hp.com/techreports/2001/HPL-2001-118.html.

METROPOLIS, N., ROSENBLUTH, A., ROSENBLUTH, M., TELLER, A., AND TELLER, E. 1953. Equations of state calculations by fast computing machines. *Journal of Chemical Physics 21*, 1087–1091.

MORRIS, R. 2002. Storage: from atoms to people. Keynote at Conference on File and Storage Technologies (FAST).

NEMHAUSER, G. AND WOLSEY, L. 1988. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD international Conference on the Management of Data*. ACM Press, Chicago, IL, 109–116.

SCHEUERMANN, P., WEIKUM, G., AND ZABBACK, P. 1998. Data partitioning and load balancing in parallel disk systems. *VLDB Journal: Very Large Data Bases 7,* 1, 48–66.

SHOR, P. 1986. The average-case analysis of some on-line algorithms for bin packing. *Combinatorica 6*, 179–200.

SHRIVER, E. 1996. A formalization of the attribute mapping problem. Technical report HPL–SSP–95–10, HP Labs. July. http://www.hpl.hp.com/SSP/papers.

SHRIVER, E. 1997. Performance modeling for realistic storage devices. Ph.D. thesis.

TOYODA, Y. 1975. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science 21*, 12, 1417–1427.

UYSAL, M. 2004. Personal communication.

UYSAL, M., ALVAREZ, G. A., AND MERCHANT, A. 2001. A Modular, Analytical Throughput Model for Modern Disk Arrays. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS 2001)*. IEEE, Cincinnati, OH.

VEITCH, A. AND KEETON, K. 2003. The Rubicon workload characterization tool. Technical report HPL–SSP–2003–13, HP Labs. March. http://www.hpl.hp.com/SSP/papers.

WARD, J., O'SULLIVAN, M., SHAHOUMIAN, T., AND WILKES, J. 2002. Appia: Automatic storage area network fabric design. In *Conference on File and Storage Technologies (FAST)*. USENIX, Monterey, CA, 203–217.

WEIKUM, G., ZABBACK, P., AND SCHEUERMANN, P. 1991. Dynamic file allocation in disk arrays. In *SIGMOD Conference*. ACM Press, Denver, CO, 406–415.

WILKES, J. 2001. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the International Workshop on Quality of Service.* Springer, Karlsruhe, Germany, 75–91.

WOLF, J. 1989. The placement optimization program: a practical solution to the disk file assignment problem. In *ACM SIGMETRICS Conference*. ACM Press, Berkeley, CA, 1–10.